KLE Society's
KLE Technological University

A Mini Project Report On

Flat buffer Conversions for Text Spell Checker

submitted in partial fulfillment of the requirement for the degree of

Bachelor of Engineering
In
Computer Science and Engineering

Submitted By

Sneha K Bankolli	01fe17bcs206
Suvrojyoti Mandal	01fe17bcs224
Swastik Tiwari	01fe17bcs226
Tanya Ejanthkar	01fe17bcs229

Under the guidance of
DR.MEENA S.M

.

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

HUBLI–580 031 (India).

Academic year 2019-20

KLE Society's
KLE Technological University

KLE Society's

KLE Technological University

2019 - 2020

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that Mini Project entitled Flat Buffer Conversions For Text Spell Checker is a bonafied work carried out by the student team Ms. Sneha K Bankolli- 01fe17bcs206, Mr.Suvrojyoti Mandal-01fe17bcs224 , Mr.Swastik Tiwari.- 01fe17bcs224, Ms.Tanya Ejanthkar - 01fe17bcs229, in partial fulfillment of completion of Fifth semester B. E. in Computer Science and Engineering during the year 2019 – 2020. The project report has been approved as it satisfies the academic requirement with respect to the project work prescribed for the above said programme.

Guide **Head of SoCSE**

Dr. Meena S. M **Dr. Meena S. M**

External Viva:

Name of the Examiners **Signature with date**

1.

2.

ABSTRACT

The objective of this project is to develop an application which efficiently suggests auto correction for the words that have been entered by the user and the ram usage for the given application should be minimized. The ram optimization is essential as the application being developed is to be deployed on embedded systems such as smart phones.

In order to store the given data in a structured way and to perform several operations such as search and insertion efficiently, data structures are used.

The data structure is used to store the dictionary of words on which search operation to find the most suitable autocorrected suggestions is the BK Tree. In order to minimize the ram usage, the dictionary is stored and accessed in the secondary storage.

The application can further be improved by applying algorithms like frequency heap which provide the autocorrected suggestions based on the frequency of the usage of the word.

.

ACKNOWLEDGEMENTS

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of a number of individuals whose professional guidance and encouragement helped us in the successful completion of this report work.

We take this opportunity to thank Dr. Ashok Shettar, Vice-chancellor, KLE Technological University and to Dr. Prakash Tewari, Principal, B V Bhoomaraddi College of Engineering and Technology, Hubli.

We also take this opportunity to thank Dr. Meena S M, Head, School of Computer Science and Engineering for having provided us academic environment which nurtured our practical skills contributing to the success of our project.

We sincerely thank Ms. Padmashree Desai, Mr. Prashant Narayankar, School of Computer Science and Engineering for their support, inspiration and wholehearted co-operation during the course of completion.

Our gratitude will not be complete without thanking the Almighty God, our beloved parents, our seniors and our friends who have been a constant source of blessings and aspirations.

1. Introduction

1.1 Overview

A data structure for embedded device needs to be developed where a user shall receive suggestions for the incorrect word typed. The data structure shall minimize the ram usage. The query time for each search must be optimized.

1.2 Motivation

The use of data structures minimizes the usage of ram and optimizes the time taken for performing the search operation whenever a user gives a word.

For example Segment trees are used for efficient range queries, union find speed up the graph connectivity algorithms, and maps and sets are used in numerous applications. As the given application being developed is to be deployed on an embedded system where the resources such as ram are limited, the ram usage should be minimized.

The existing approaches to the given problem statement either don't efficiently use the ram or are time taken for each search performed is not optimized.

One of the existing solutions to the problem statement is sym-spell which creates a key for each word and stores it in memory, which is quite fast but takes a lot of storage.

Therefore the main aim for the solution that is being developed shall reduce the memory usage due to the limited ram available on embedded devices and also to improve the time taken for each search.

1.3 Objectives of the Project

- Experiment and analyze different approaches to optimize memory used in primary and secondary storage along with response time.
- Find auto-correct suggestions based on suitable ranking criteria.
- Develop an android spell checker application that stores the data structure as a flat buffer binary.

1.4 Literature Survey

A basic spell checker carries out the following processes:
It scans the text and extracts the words contained in it. It then compares each word with a known list of correctly spelled words (i.e. a dictionary). This might contain just a list of words, or it might also contain additional information, such as hyphenation points or lexical and grammatical attributes.

An additional step is a language-dependent algorithm for handling morphology. Even for a lightly inflected language like English, the spell-checker will need to consider different forms of the same word, such as plurals, verbal forms, contractions, and possessives. For many other languages, such as those featuring agglutination and more complex declension and conjugation, this part of the process is more complicated.

It is unclear whether morphological analysis—allowing for many different forms of a word depending on its grammatical role—provides a significant benefit for English, though its benefits for highly synthetic languages such as German, Hungarian or Turkish are clear.

As an adjunct to these components, the program's user interface will allow users to approve or reject replacements and modify the program's operation.

An alternative type of spell checker uses solely statistical information, such as n-grams, to recognize errors instead of correctly-spelled words. This approach usually requires a lot of effort to obtain sufficient statistical information. Key advantages include needing less runtime storage and the ability to correct errors in words that are not included in a dictionary.

In some cases spell checkers use a fixed list of misspellings and suggestions for those misspellings; this less flexible approach is often used in paper-based correction methods, such as the see also entries of encyclopedias.

Clustering algorithms have also been used for spell checking combined with phonetic information. But the most popular approach to find the closest word is calculating levenshtein distance with all the words in the dictionary and displaying the words with minimum levenshtein distance. Levenshtein distance of two words is the number of insertions/deletions/changes required in first word to convert it to second word
There are three different levenshtein distances**:**

- Levenshtein distance: adjacent transposition (AC->CA) counted as 2 edits. The triangle inequality does hold**.**
- Restricted Damerau-Levenshtein distance (Optimal string alignment algorithm): adjacent transposition counted as 1 edit, but substrings can't be edited more than once: ed("CA" , "ABC") =3. The triangle inequality does not hold**.**
- True Damerau-Levenshtein distance: adjacent transposition counted as 1 edit, substrings can be edited more than once: ed("CA" , "ABC") =2. The triangle inequality does hold**.**

The four different algorithms that were compared and benchmarked are:

- Norvig's Spelling Corrector
- BK-tree (Burkhard-Keller-tree)
- SymSpell (Symmetric Delete spelling correction algorithm)
- LinSpell (Linear search spelling correction algorithm)

Norvig's Spelling Corrector

Norvig's algorithm is using the true Damerau-Levenshtein edit distance. It could be modified to use the Levenshtein distance. The idea is if we artificially generate all terms within maximum edit distance from the misspelled term, then the correct term must be among them. We have to look all of them up in the dictionary until we have a match. So all possible combinations of the 4 spelling error types (insert, delete, replace and adjacent switch) are generated. This is quite expensive with e.g. 114,324 candidate term generated for a word of length=9 and edit distance=2.

BK-tree

The BK-tree utilizes the triangle inequality, a property of the Levenshtein edit distance:
Levenstein(A,B)+Levenstein(A,C)≥Levenstein(B,C) and
Levenstein(A,B)−Levenstein(A,C)≤Levenstein(B,C).

During indexing the Levenshtein(root node,child node) are precalculated. During lookup we calculate Levenshtein(input ,root node). The triangle inequality is used as a filter, to only recursively follow those child nodes where the precalculated Levenstein(root node,child node) is in the range [Levenstein(input ,root node)-dmax, Levenstein(input ,root node)+dmax].

SymSpell Algorithm

SymsSpell is an algorithm to find all strings within an maximum edit distance from a huge list of strings in very short time. It can be used for spelling correction and fuzzy string search. SymSpell derives its speed from the Symmetric Delete spelling correction algorithm and keeps its memory requirement in check by prefix indexing.

The Symmetric Delete spelling correction algorithm reduces the complexity of edit candidate generation and dictionary lookup for a given Damerau-Levenshtein distance. It is six orders of magnitude faster (than the standard approach with deletes + transposes + replaces + inserts) and language independent.

Opposite to other algorithms only deletes are required, no transposes + replaces + inserts. Transposes + replaces + inserts of the input term are transformed into deletes of the dictionary term. Replaces and inserts are expensive and language dependent: e.g. Chinese has 70,000 Unicode Han characters!

The speed comes from inexpensive delete-only edit candidate generation and pre-calculation. An average 5 letter word has about 3 million possible spelling errors within a maximum edit distance of 3, but SymSpell needs to generate only 25 deletes to cover them all, both at pre-calculation and at lookup time. The idea behind prefix indexing is that the discriminatory power of additional chars is decreasing with word length. Thus by restricting the delete candidate generation to the prefix, we can save space, without sacrificing filter efficiency too much. In the benchmark three different prefix lengths lp=5, lp=6 and lp=7 were used. They reflect different compromises between search speed and index size. Longer prefix length means higher search speed at the cost of higher index size.

LinSpell

This is basically a linear scan through the word list and calculating the edit distance for every single word (with a few tweaks). It was intended as the baseline measure for the benchmark. Surprisingly enough and despite its O(n) characteristics it turned out to excel both BK-tree and Norvig's algorithm.

There are several reasons for that:

- do not underestimate the constants in the Big O notation. Visiting only 20% of the nodes in a BK-tree is more expensive than a linear search where the atomic cost is only 10%.

- As Damerau-Levenshtein calculation is very expensive it is not the number of processed words which matters, but the number of words where we need the full Damerau-Levenshtein calculation. We can speedup the search if we can prefilter words from the calculation or terminate the calculation once a certain edit distance is reached.

- If we restrict the search to best match we can utilize options for early termination of the search.

- words with no spelling errors are a frequent case. Then the lookup can be done with a hash table or trie in O(1) ! If we restrict the search to best match, we can instantaneously terminate the search.

- We do not need to calculate the edit distance if Abs(word.Length-input.Length) > Maximum edit distance (or best edit distance found so far if we restrict the search to best match)

- If we restrict the search to best match, we can terminate the edit distance calculation once the best edit distance found so far is reached.

- If we restrict the search to best match, and we have found already one match with edit distance=1, then we do not need to calculate the edit distance if the count of the term in question is smaller than the count of the match already found.

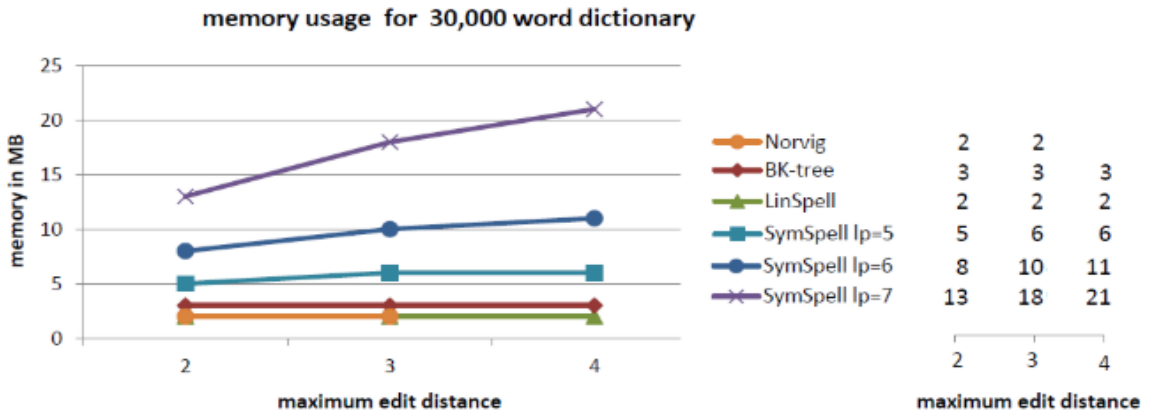


Figure 1: Memory in MB vs edit distance graph

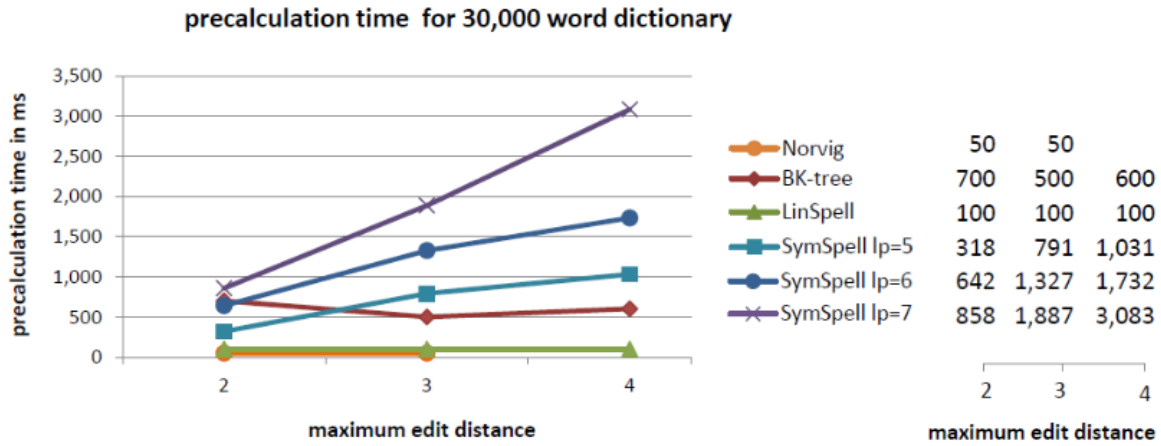


Figure 2: Precalculation time in ms vs Edit distance graph

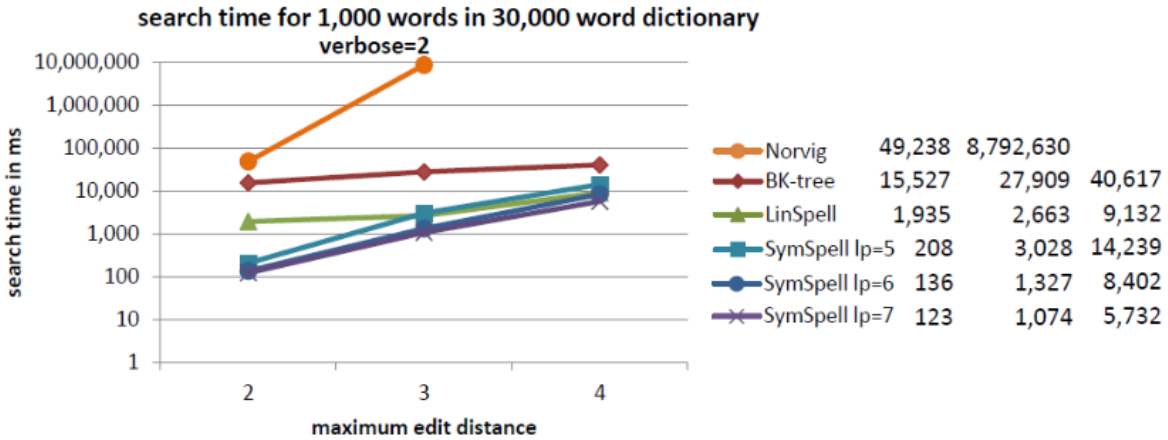


Figure 3: Search time in ms vs edit distance graph

1.5 Problem Definition

Develop an appropriate data structure that performs spell checking and suggests autocorrect suggestions and needs to be integrated with embedded devices.

2. Proposed System

2.1 Description of Proposed System.

A data structure needs to be developed where a user will receive autocorrect suggestions for the words typed. The ram usage shall be 20mb or less. The secondary memory usage should be 100mb or less

2.2 Description of Target Users

Users can use the application to easily type content without making spelling errors as they will be aided by the autocorrect feature of the application.

2.3 Advantages/Applications of Proposed System

It can be used any time even without an internet access. It will find any spelling mistake, while you are typing, and offer suggestions from its dictionary. With the ease of typing on computers, people are typically able to write more text faster than they would by hand or on a typewriter.

2.4 Scope

At present the autocorrect program consumes a ram of 149kb and a secondary storage of 49mb. The memory consumption can be tried to be reduced further to improve performance.

The time taken to perform a search operation to get the autocorrect suggestions on 10000 is 6 seconds. This can be improved further by applying algorithms like frequency heap.

1. Software Requirement Specification

3.1 Overview of SRS

A software requirements specification (SRS) is a description of a software system to be developed. It lays out functional and nonfunctional requirements and may include a set of use cases that describe user interactions that the software must provide.[2]

3.2 Requirement Specifications

3.2.1 Functional Requirements

a. A user shall receive autocorrect suggestions when a word is typed in the application.
b. Feature of insertion of the word in the BK Tree should be for the developer.

3.2.2 Use case Diagrams

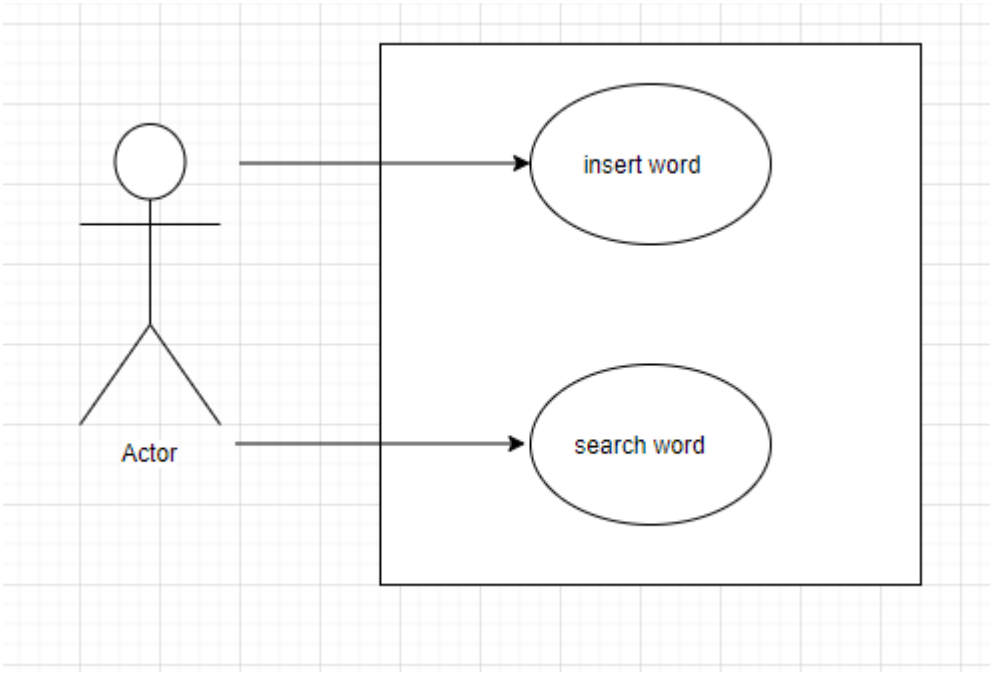


Figure 4: Use case diagram

3.2.3 Use Case Descriptions using Scenarios

Use Case: insert word
Actors: User
Goal: to insert word.
Pre-conditions: BK tree available in memory
Post-conditions: the word was inserted in BK if insertion is successful
Success Scenario: word successfully inserted into BK tree.
Exception: word contains characters other than alphabets

Table No: 3.2.3(a)

Use Case: search word
Actors: User
Goal: to find suggestions.
Pre-conditions: BK tree available in memory
Post-conditions: Suggestion output
Success Scenario: Successful search autocorrect suggestion within TOL edit distance
Exception: No valid suggestion found within TOL edit distance

Table No: 3.2.3(b)

3.2.4 Nonfunctional Requirements

a. The data structure should be deployed on an embedded device application.
b. The ram usage should be 20mb or less.
c. The secondary memory usage should be 100mb or less.
d. The time taken should be 12 seconds or less for dictionary of size of 10000 words

4. System Design

4.1 Architecture of the System

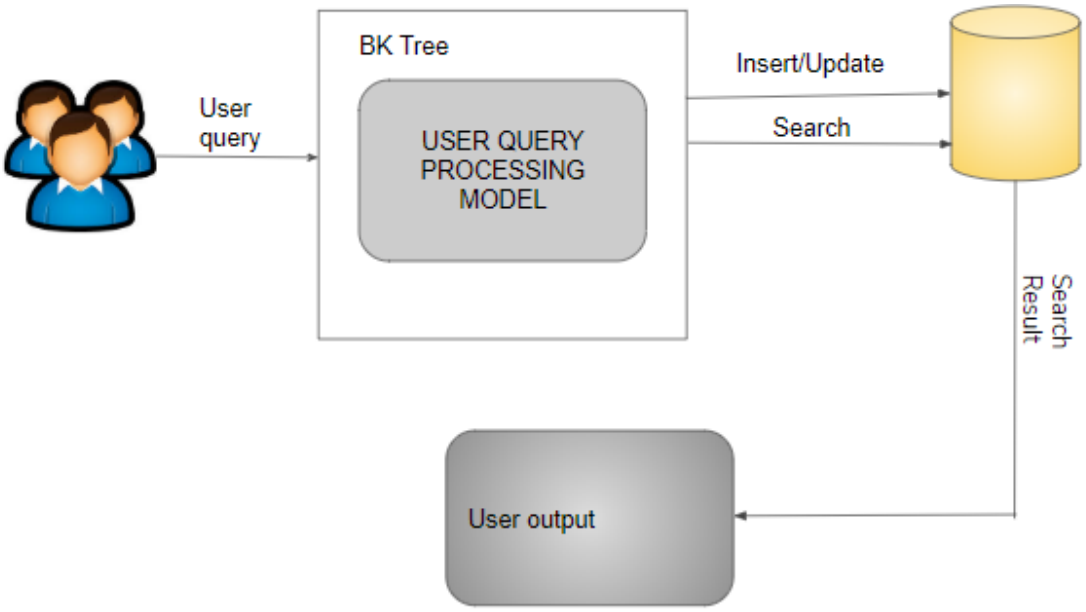


Figure 5: System Architecture

4.1 Level 0 DFD

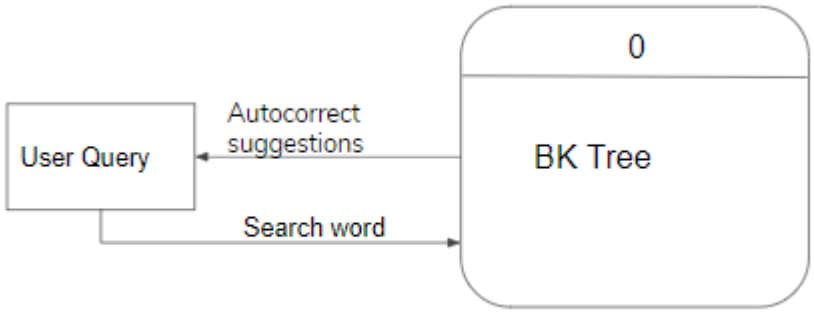


Figure 6: Data Flow Diagram Level 0

4.2 Detailed DFD for the Proposed System

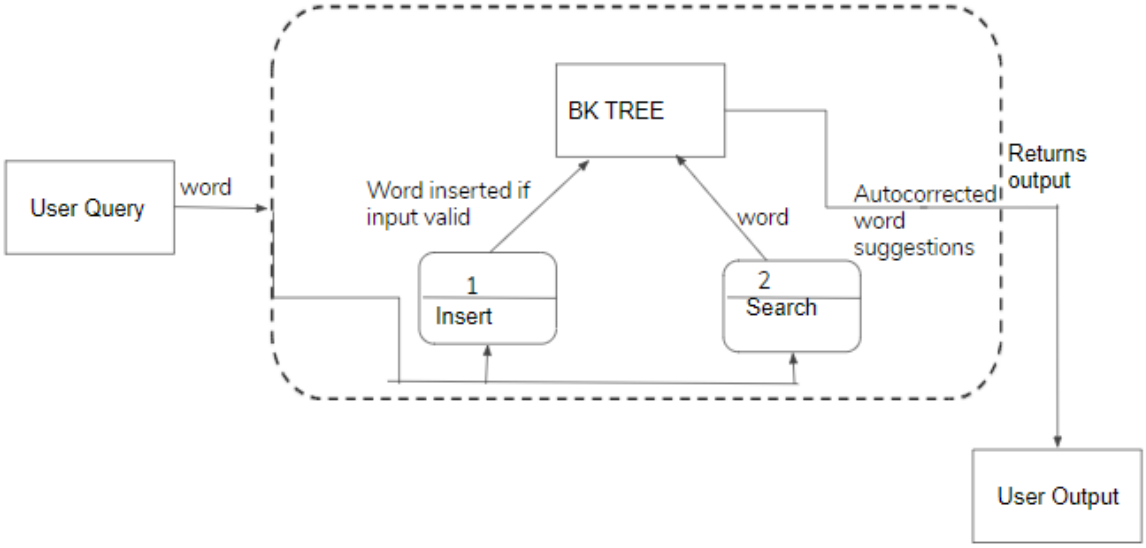


Figure 7: Detailed Data Flow Diagram

4.3 Sequence Diagram

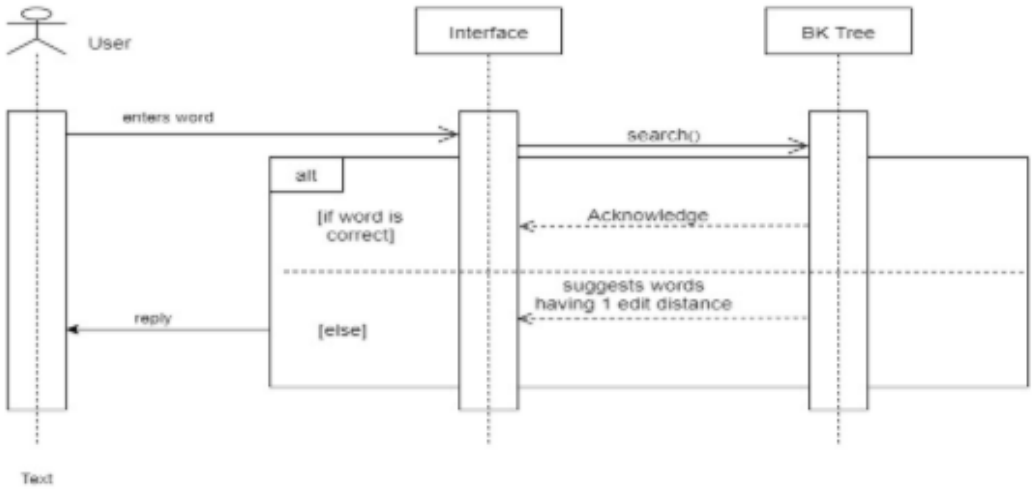


Figure 8: Sequence Diagram

4.4 Data Structure used

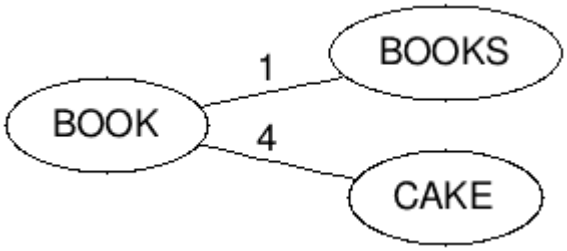
Burkhard Keller Tree:

Proposed by Burkhard and Keller in 1973, the BK-Tree is a data structure used for spell checking based on the Levenshtein Distance between two words, which is basically the number of changes you need to make to a word to turn it into another word.
Some example distances:
LevenshteinDistance(cook, book) -> 1
LevenshteinDistance(cook, books) -> 2
LevenshteinDistance(what, water) -> 3

To make the last example clear, the distance between what and water is based on three moves, the first is to drop the h, then add the e and finally add the r. You can use this distance to work out how close someone is to spelling a word correctly, for instance if I want to know out of the set of words [cook, book, books, what, water] which word the misspelling wat is closest to I could do a full scan of the list and find the words with the lowest distance:
LevenshteinDistance(wat, cook) -> 4
LevenshteinDistance(wat, book) -> 4
LevenshteinDistance(wat, books) -> 5
LevenshteinDistance(wat, what) -> 1
LevenshteinDistance(wat, water) -> 2

Based on that search I can determine the user probably meant the word what due to it having the lowest distance of 1.
This works in the case of a small number of words since O(n) isn't bad in this case, however if I wanted to scan a full dictionary for the closest match O(n) isn't optimal. This is where a BK-Tree comes in.

To build a BK-Tree all you have to do is take any word from your set and plop it in as your root node, and then add words to the tree based on their distance to the root. For instance if I started a tree with the word set [book, books, cake] then my tree would look like this if I started by making the word book my root node:

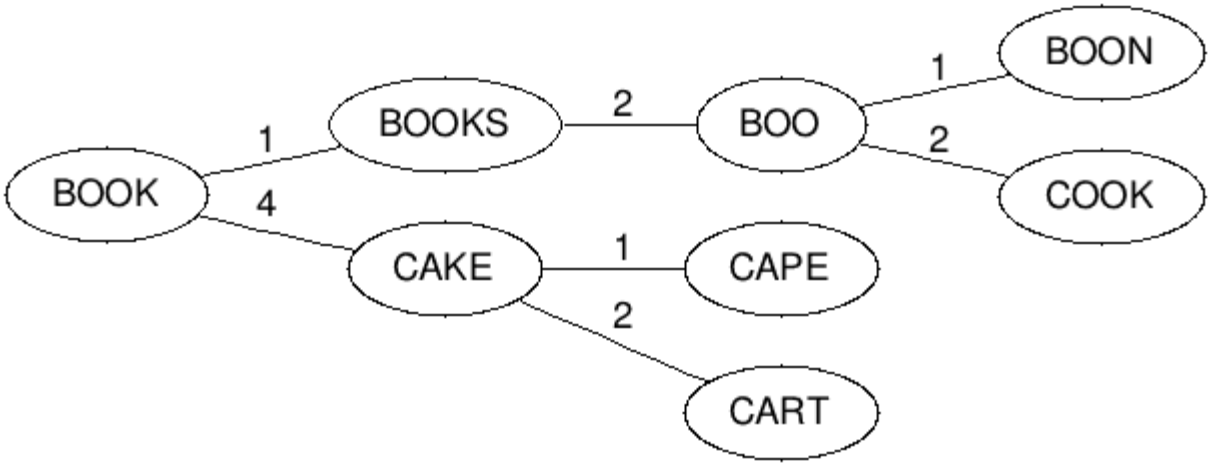


This is because:

LevenshteinDistance(book, books) -> 1
LevenshteinDistance(book, *cake*) -> 4

Of course now if we add the word boo to the mix we get a conflict because:
LevenshteinDistance(boo*k*, boo) -> 1
which collides with books. To handle this we now have to chain boo from books by making it a child of books based on their Levenshtein Distance.
LevenshteinDistance(boo*ks*, boo) -> 2

We continue to use this strategy as we add words, so if we throw in [Cape, Boon, Cook, Cart] then we get this:



Now that we have our structure the obvious question is how do we search it? This is simple as now all we need to do is take our misspelled word and find matches within a certain level of tolerance, which we'll call N. We do this by taking the Levenshtein Distance of our word and compare it to the root, then crawl all nodes that are that distance ±N.
For example, if we searched for the closest match to the misspelling caqe within the tolerance of 1 in the above structure we would crawl it like this:

1) LevenshteinDistance(caqe, book) -> 4
   a) Check (4 <= 1) - Don't add book as a match
   b) Crawl all edges between 3 and 5 (4-1,4+1)
2) Crawl into edge path 4 from book
3) LevenshteinDistance(caqe, cake) -> 1
   a) Check (1 <= 1) - Add cake as a match
   b) Crawl all edges between 0 and 2 (1-1, 1+1)
4) Crawl into edge path 1 from cake
5) LevenshteinDistance(caqe, cape) -> 1
   a) (1 <= 1) - Add cape as a match
6) Crawl into edge path 2 from cake
7) LevenshteinDistance(caqe, cart) -> 2

a) Check (2 <= 1) - Don't add cart as a match

From this example it appears we can now find misspellings at a O(log n) time, which is better than our O(n) from before. This however doesn't tell the whole story, because our tolerance can drastically increase the number of nodes we need to visit in order to do the search.

4.5 Data Set Description

There are 10,000 given words in the dictionary from which the word entered by user has to be searched.

5. Implementation

5.1 Proposed Methodology

The proposed solution consists of the following modules:

Spell Check: The correctness of the spelling is judged by checking its existence in the bk tree.

Data Structure to find the closest spelling: A BK-tree is implemented with the existing words and in case the user entered an incorrect word, a query will be performed in BK tree, to find the closest matching words.

Checking the correctness of the spelling by searching in the bk tree whether the word exists or not.

- Time Complexity
  - Average Case Θ(length of searched word)
  - Worst case O(26* (length of searched word))
- Space Complexity
  - O (sum of length of all words stored in the dictionary)

5.2 Description of Modules

- **Module Name:** search ()

This module searches the BK Tree to get the autocorrected suggestions with edit distance within TOL.

Input: User entered word.

Output: Autocorrected suggestions or message if no valid suggestions found.

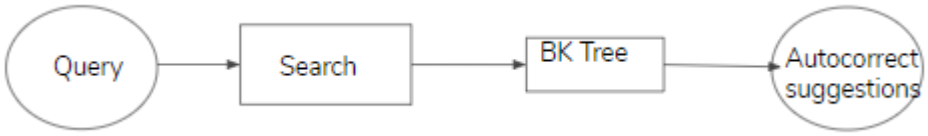


Figure 9: Search Module

- **Module Name:** insert ()

This module is used to insert a new word in the BK tree.

Input: Word to be inserted

Output: Message displaying if insertion in BK tree successful or not

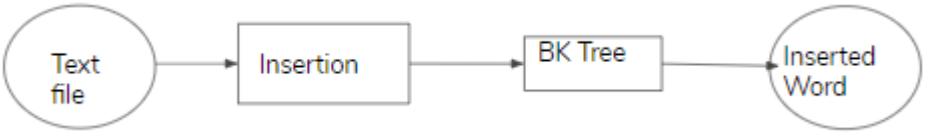


Figure 10: Insert Module

6. Testing

6.1 Acceptance Test Plan

Required id	Test id	Input Description	Expected Output	Actual Output
3.1	1.	Valid word	word	word
3.2	2.	Valid prefix	Autocompleted word	Autocompleted word
3.3	3.	Word	autocorrected suggestions	autocorrected suggestions
3.4	4	Word	No suggestions found	No suggestions found

6.2 Unit Test Plan:

Module: Unit test plan for module search ()

Required id	Test id	Input Description	Expected Output	Expected Output
3.1	1.	Valid Word eg:hello	Same word displayed hello	Same word displayed hello
3.2	2.	Valid prefix eg:coo	Autocompleted Word cook	Autocompleted Word cook
3.3	3.	Word not found eg:mook	autocorrected suggestions cook look	autocorrected suggestions cook look
3.4	4.	Word (with more than TOL edit distance)	No suggestions found	No suggestions found

Module: Unit test plan for module insert ()

Required id	Test id	Input Description	Expected Output	Expected Output
3.5	1.	Word to be inserted eg: happy	Word successfully inserted	Word successfully inserted
3.6	2.	Word to be inserted eg:cook	Word already inserted	Word already inserted
3.7	3.	Word to be inserted eg: happy	Word already inserted	Word already inserted

7. Results and Analysis

```
Enter 1 for suggestions 2 for exit
1
Enter word
hit
[tit, it, hot, ait, hid, nit, hat, his, hut, bit, kit, hit]
```

Figure 11: Search result for 'hit'

When a word is entered all the words within 1 edit distance are displayed

```
Enter 1 for suggestions 2 for exit
1
Enter word
kjxs
[]
```

Figure 12: Search result for 'kjxs'

When a word is entered for which there are no words within 1 edit distance no words are displayed.

```
(base) swastik@pop-os:~/Downloads$ cd "/home/swastik/Downloads/" && g++ bk
loads/"bkgtree copy
Time Taken in nanoseconds for 10000 words
6032485
(base) swastik@pop-os:~/Downloads$
```

Figure 13: Time taken for finding similar words for 10000 words

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	2,216,589	72,712	72,704	8	0
2	2,339,422	73,216	73,192	24	0
3	2,422,498	83,520	83,464	56	0
4	2,469,293	83,520	83,464	56	0
5	2,546,071	83,520	83,464	56	0
6	2,695,352	83,560	83,496	64	0
7	2,741,593	83,560	83,496	64	0
8	2,842,705	83,560	83,496	64	0
9	2,963,574	83,560	83,496	64	0

Figure 14: Actual time and space complexities in system

For a dictionary of 80000 words
RAM USAGE: 84 KB
ROM USAGE: 46MB
Time Consumed: 6 seconds for 10000 suggestions

8. Conclusions And Future Scope

At present the autocorrect program consumes a ram of 149kb and a secondary storage of 49mb. This memory consumption can be tried be reduced further to improve performance.
The time taken to perform a search operation to get the autocorrect suggestions on 10000 words is 6 seconds.

The data structure needs to be integrated with an embedded device application and suitable ranking algorithm for top k suggestions can be developed.

9. Bibliography

- https://github.com/cathtunti/CS51--SpellChecker/blob/master/report.pdf
- https://towardsdatascience.com/symspell-vs-bk-tree-100x-faster-fuzzy-string-search-spell-checking-c4f10d80a078
- https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/
- https://norvig.com/spell-correct.html

10. Gantt Chart